

## ÍNDICE

RESUMEN .....	2
1. PLAN DE REMOCION DE DEFECTOS .....	2
2. REVISION DE LOS REQUERIMIENTOS .....	4
3. REVISION DEL DISEÑO .....	4
4. REVISION DEL CODIGO.....	4
5. ANALISIS ESTATICO DE CODIGO .....	5
6. PRUEBAS UNITARIAS .....	7
7. PRUEBAS DE SISTEMA.....	9
8. COSTOS DE REMOCION DE DEFECTOS .....	9
9. INTEGRACION CONTINUA .....	9
CONCLUSIONES .....	10

## RESUMEN

El objetivo general del presente trabajo es aplicar los conceptos de Calidad de Software aprendidos durante el transcurso del semestre. En el siguiente informe se presentan las diferentes implementaciones prácticas de los conceptos aplicados al trabajo final de la materia Ingeniería de Software.

### Metodología de Trabajo

A los fines de cumplir con los objetivos deseados, el modo de operar consta en una fuerte investigación en la web de las herramientas necesarias. Seguida de la implementación de las mismas desde diferentes acercamientos, procurando siempre el consenso grupal.

Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una comparación de lo logrado con resultados que se habían obtenido previamente en el trabajo final de la materia Ingeniería de Software.

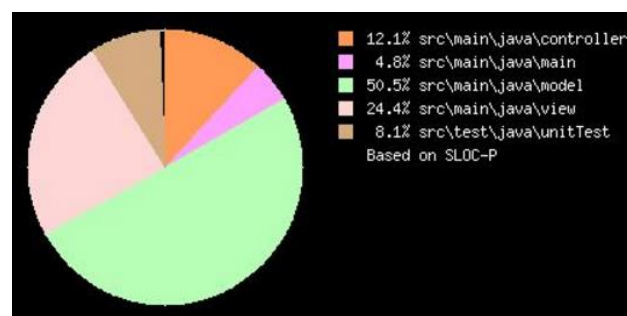
## 1. PLAN DE REMOCION DE DEFECTOS

Utilizando la herramienta **LocMetrics**, determinamos que el trabajo final de Ingeniería de Software tiene un total de 1461 líneas de código.

Progress			
Source Files	26	C&SLOC, Code & Comment	0
Directories	54	CLOC, Comment Lines	19
LOC, Lines of Code	1461	CWORD, Comment Words	52
BLOC, Blank Lines	321	HCLOC, Header Comments	0
SLOC-P, Executable Physical	1121	HCWORD, Header Words	0
SLOC-L, Executable Logical	858		
McCabe VG Complexity	81		

Del total, existen 321 líneas que están en blanco y 19 se corresponden a líneas de comentarios puros, lo que deja un total de 1121 líneas de código ejecutables.

Estas se encuentran distribuidas dentro del proyecto de la siguiente manera:



Utilizando la definición de Capers Jones, para estimar la cantidad de defectos esperables en el proyecto: “1 por cada 10 oportunidades”. Podemos decir que de las 1121 de código ejecutable se espera encontrar al menos 112 defectos.

Lo cual podría resultar un poco exagerado, para mejorar la estimación utilizamos valores históricos de tamaños similares de inspecciones de código realizadas entre los miembros del grupo e ingeniería inversa de los defectos reales luego del release, determinando un total de 47 defectos esperables dentro del proyecto.

Estimando encontrar los mismos distribuidos de la siguiente manera en cada etapa:

- **Especificación de Requerimientos:** 7
- **Diseño:** 16
- **Codificación Unitaria:** 14
- **Codificación de la Integración:** 5
- **Documentación:** 5
- **Pruebas de Sistema:** 0
- **Software en Operación:** 0

La Cessi, Cámara de la Industria Argentina del Software, reveló los resultados de su encuesta, realizada a 150 empresas de todo el país y con el único objetivo de conocer cuánto pagan a sus más de 12.000 desarrolladores de software.<sup>1</sup>

Los datos obtenidos, arrojan un salario bruto mensual promedio de \$13.100 para la categoría de programadores sin experiencia previa (Junior), \$17.598 con alguna experiencia (Semisenior) y \$23.500 para los programadores con experiencia (Senior).

Teniendo en cuenta que el proyecto fue desarrollado por estudiantes, es decir, programadores sin experiencia procedemos a calcular el costo por hora de trabajo. Teniendo en cuenta una jornada laboral completa de 8hs, 5 días a la semana, 4 semanas al mes. Obtenemos un total de 160 hs/mes. Por lo tanto \$13.100/160hs arroja un precio por hora de \$82 la hora de trabajo.

Comparando este valor con la cantidad de errores encontrados al momento de la entregar el release en la materia ingeniería de software (apenas 3), podemos percibir que probablemente existan una gran cantidad de defectos aun latentes sin descubrir y/o corregir.



Esto con seguridad implicara mayores costos a la hora de corregir los defectos.

**NOTA:** El detalle del plan se encuentra anexado bajo el nombre PlanRemocionDefectos.xls

## 2. REVISION DE LOS REQUERIMIENTOS

La validación de requerimientos trata de mostrar que estos realmente definen el sistema que el cliente desea. Es importante debido a errores en el documento de requerimientos pueden conducir a importantes costos al repetir el trabajo cuando son descubiertos durante el desarrollo o después de que el sistema esté en uso.

No encontramos que falten o sobren requerimientos, sin embargo algunos de estos no se encuentran bien redactados. Se generó un reporte de revisión para que los mismos sean corregidos.

En esta etapa encontramos una cantidad similar de defectos a la esperada de acuerdo al plan de remoción de defectos.

**NOTA:** El documento de revisión se encuentra anexado bajo el nombre RevisionDeRequerimientos.xls

## 3. REVISION DEL DISEÑO

Como algunos miembros del trabajo actual no se encontraban al momento de generar los documentos de diseño se genera una reunión para validar el diseño conceptual. Nos aseguramos de que todos los aspectos relativos a los requerimientos han sido apropiadamente contemplados en el diseño.

Durante la revisión se presentó a los otros miembros el diseño conceptual. Al hacerlo, se demuestro que el sistema tiene la estructura requerida, las funciones y las características especificadas por los documentos de análisis.

Todos los participantes, en conjunto, verificamos que el diseño propuesto estaba correcto. No encontramos la cantidad de defectos esperada por el plan de remoción de defectos. Probablemente esto se deba a que fue uno de los temas en los que más énfasis se hizo en el trabajo anterior (Ing. de Software).

**NOTA:** El documento de revisión se encuentra anexado bajo el nombre RevisionDeDiseno.xls

## 4. REVISION DEL CODIGO

Con el objetivo de mejorar la calidad del software y detectar errores de manera temprana, se realizaron revisiones de código informales por parte de los integrantes del grupo que no participaron en la codificación durante la materia Ingeniería de Software (por cursar en años distintos).

Estos generaron los comentarios, y posteriormente todos juntos determinamos mediante discusión abierta las posibles mejoras en el producto.

**NOTA:** El documento de inspección se encuentra anexado bajo el nombre ReporteDeInspeccion.xls

## 5. ANALISIS ESTATICO DE CODIGO

### CHECK STYLE

Descripción: Este plugin implementa una herramienta de desarrollo para ayudar a los programadores escribir código Java que se adhiere a un estándar de codificación. Automatiza el proceso de verificación de código, y nos libera de esta aburrida (pero importante) tarea. Es ideal para proyectos que quieren hacer cumplir un estándar de codificación.

Reporte:

Summary	
Files	Errors
23	2081
Files	
Name	Errors
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	361
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BeattModel.java	337
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\view\DJView.java	337
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\HeartModel.java	219
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\HeartAdapter.java	105
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletAdapter.java	102
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\controller\BulletController.java	88

Resultado y decisión: Se observaron una gran cantidad de defectos de estilo en el reporte, ya que este está validando nuestro código con el estándar sugerido por Google<sup>2</sup>. Esto se debe a que inicialmente cuando se desarrolló el código en Ingeniería de Software, no se optó por seguir un estándar de codificación.

La corrección de estos defectos probablemente demande una gran cantidad de tiempo y no produzca grandes beneficios para el cliente. Por lo tanto se optó por hacer caso omiso a los mismos.

### PMD

Descripción: Este plugin es un analizador de código fuente. A través del cual se encuentra defectos habituales de programación como las variables inutilizadas, bloques catch vacíos, la creación de objetos innecesarios, y así sucesivamente. Es compatible con Java, JavaScript, Salesforce.com Apex, PLSQL, Apache Velocity, XML, XSL.

Reporte:

**Problems found**

#	File	Line	Problem
1	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	40	<a href="#">Useless parentheses.</a>
2	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	48	<a href="#">Useless parentheses.</a>
3	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	50	<a href="#">Useless parentheses.</a>
4	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	58	<a href="#">Useless parentheses.</a>
5	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	69	<a href="#">Avoid empty catch blocks</a>
6	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	105	<a href="#">Useless parentheses.</a>
7	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java	105	<a href="#">Useless parentheses.</a>
8	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\HeartModel.java	55	<a href="#">Avoid empty catch blocks</a>
9	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\view\BeatBar.java	24	<a href="#">Avoid empty catch blocks</a>
10	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\view\BeatBar.java	26	<a href="#">An empty statement (semicolon) not part of a loop</a>
11	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\view\Screen.java	29	<a href="#">Avoid empty catch blocks</a>
12	C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\view\Screen.java	31	<a href="#">An empty statement (semicolon) not part of a loop</a>

**Resultado y decisión:** Este reporte nos arrojó varias advertencias sobre uso de paréntesis innecesarios, bloques catch vacíos, variables no inicializadas, etc.

Si bien algunos defectos parecen triviales hay algunos otros que podrían haber derivado en fallas durante la ejecución del software. Como el costo de corregir estos defectos es mínimo, se decide de forma unánime hacer las correcciones sugeridas por la herramienta.

**CPD**

**Descripción:** Este plugin ofrece un mecanismo de detección automática de copy/paste de líneas de código dentro del proyecto. Es ideal para mantener un control automático sobre esta práctica habitual pero no recomendada (reutilizar bloques de código).

**Reporte:**

```
<pmd-cpd>
  <duplication lines="55" tokens="268">
    <file line="107" path="C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\BulletModel.java"/>
    <file line="62" path="C:\Users\bersu\Desktop\IngSoft-2016-NullSoft\src\main\java\model\HeartModel.java"/>
    <codefragment>
      <![CDATA[
        } public void registerObserver(BeatObserver o) { beatObservers.add(o); } public void removeObserver(BeatObserver o) { int i
        public void notifyBeatObservers() { for (int i = 0; i < beatObservers.size(); i++) { BeatObserver observer = (BeatObserver) be
        registerObserver(BPMObserver o) { bpmObservers.add(o); } public void removeObserver(BPMObserver o) { int i = bpmObservers.inc
        notifyBPMObservers() { for (int i = 0; i < bpmObservers.size(); i++) { BPMObserver observer = (BPMObserver) bpmObservers.get(i
        { bulletObservers.add(o); } public void removeObserver(BulletObserver o) { int i = bulletObservers.indexOf(o); if (i >= 0) {
        (int i = 0; i < bulletObservers.size(); i++) { BulletObserver observer = (BulletObserver) bulletObservers.get(i); observer.upc
        ]}]
      </codefragment>
    </duplication>
  </pmd-cpd>
```

**Resultado y decisión:** Recibimos una cantidad media de advertencias de bloques duplicados dentro del código, sin embargo muchos de estos bloques pertenecen al código original propuesto del BeatModel.

Si bien es una mala práctica, y deberíamos no tener bloques duplicados. La corrección nos va a demandar un tiempo significativo, ya que deberíamos crear nuevas funciones y reestructurar el código. Se opta por hacer caso omiso a este reporte.

**FINDBUGS**

**Descripción:** Este plugin utiliza el análisis estático para buscar errores en el código Java basándose en patrones o firmas de errores típicos conocidos.

## Reporte:

### Metrics

884 lines of code analyzed, in 27 classes, in 4 packages.

Metric	Total	Density*
High Priority Warnings	7	7.92
Medium Priority Warnings	5	5.66
Low Priority Warnings	17	19.23
<b>Total Warnings</b>	<b>29</b>	<b>32.81</b>

(\* Defects per Thousand lines of non-commenting source statements)

### Summary

Warning Type	Number
<a href="#">Bad practice Warnings</a>	7
<a href="#">Multithreaded correctness Warnings</a>	3
<a href="#">Performance Warnings</a>	4
<a href="#">Dodgy code Warnings</a>	15
<b>Total</b>	<b>29</b>

**Resultado y decisión:** Esta herramienta detecto 29 defectos dentro de nuestro código, de los cuales algunos refieren a malas prácticas, otros a problemas de performance o de concurrencia.

Como el tiempo para realizar las correcciones no es grande y el impacto que podría producir en el software en producción es significativo se optó por corregir la mayor cantidad posible de los mismos.

## 6. PRUEBAS UNITARIAS







En esta, por ser una de las etapas más avanzadas se espera encontrar una cantidad reducida de defectos. La idea es utilizar pruebas unitarias incrementales, que aseguren la funcionalidad de diferentes partes del código.

### JACOCO

**Descripción:** Es una herramienta libre (GPL) escrita en Java, que nos permite comprobar el porcentaje de código al que accedemos desde los test. Es decir, nos permite saber cuánto código estamos realmente probando con nuestros test.

Además JaCoCo también nos indica la complejidad ciclomática de McCabe<sup>3</sup>. Esto nos dice como de “complejo” es un método. Esto nos puede servir para orientar nuestros test y probar primero las piezas más complejas, o incluso nos puede hacer plantearnos una refactorización para bajar la complejidad del código.

#### Reporte:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
model		62%		43%	81	140	137	315	48	96	0	5
view		65%		38%	26	42	68	171	15	29	2	8
controller		63%		n/a	15	24	33	72	15	24	0	3
main		0%		n/a	8	8	20	20	8	8	4	4
Total	850 of 2.203	61%	66 of 114	42%	130	214	258	578	86	157	6	20

De este reporte podemos distinguir claramente 3 conceptos diferentes:

- **Line Coverage:** Esta es la más simple de las métricas que nos ofrecen las herramientas de análisis de cobertura, ya que, solamente mide si una determinada línea de código se ejecuta o no.
- **Instruction Coverage:** La cobertura de instrucciones es una medida un poco más específica, ya que, considera si se incluyen múltiples instrucciones en una sola línea de código.

Es muy recomendable alcanzar una elevada cobertura de sentencias, aunque no siempre es posible por premura de tiempo o medios.

Aun habiendo conseguido una cobertura elevada de sentencias, puede ser que nos estemos engañando en las ramas condicionales.

- **Branch Coverage:** La cobertura de ramas mide la fracción de segmentos de código independientes que fueron ejecutados. Los segmentos de código independientes son secciones de código que no tienen ramas dentro o fuera de ellos. Dicho de otra manera, un segmento de código independiente es una sección de código que se puede esperar para ejecutar en su totalidad cada vez que se ejecute.

Se habla de una cobertura de ramas al 100% cuando se ha recorrido todas y cada una de las posibles vías de ejecución controladas por condiciones.

En nuestro proyecto se tiene un nivel de cobertura de ramas del 42%. A partir del reporte generado por **LocMetrics** sabemos que la complejidad ciclomatica de McCabe en nuestro proyecto es de 82. Es decir que para abarcar el 100% de cobertura de ramas deberíamos tener 82 casos de pruebas unitarias, de los cuales actualmente solo se han escrito 6.

La cobertura de ramas es indiscutiblemente deseable; pero habitualmente es un objetivo excesivamente costoso de alcanzar en su plenitud.



No hay que buscar la calidad perfecta ni el 100% de cobertura, esto no es inteligente ni práctico, ya que nos llevaría demasiado tiempo y esfuerzo. Pero si son necesario unos mínimos de calidad y enfocar nuestros esfuerzos a probar las piezas más complicadas o más importantes para negocio.

Sin medir, es imposible mejorar. Hay que medir antes y después, y comparar las medidas. Eso es lo que realmente nos indica si estamos mejorando o empeorando.

## 7. PRUEBAS DE SISTEMA

Como en el proyecto de Ingeniería de Software por falta de tiempo no se desarrollaron pruebas de sistema, va a ser necesario redactar de cero los diferentes escenarios posibles y si se tiene disponibilidad estas pruebas se automatizaran con alguna herramienta.

## 8. COSTOS DE REMOCION DE DEFECTOS

Teniendo en cuenta estas consideraciones, si el precio de la hora de trabajo es \$100. El costo de remover todos los defectos con el software en Operación asciende a \$517.000. Mientras que si se aplican técnicas para encontrar y remover defectos durante el desarrollo del proyecto, solo \$35.948

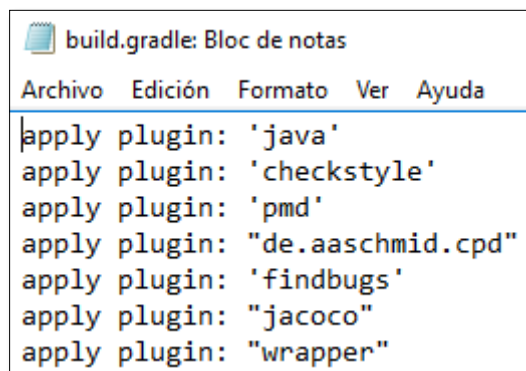
La diferencia es realmente significativa, y evidencia la importancia de detectar y corregir defectos de manera eficiente y temprana.

**NOTA:** El detalle del plan se encuentra anexado bajo el nombre PlanRemocionDefectosTarde.xls

## 9. INTEGRACION CONTINUA

Como herramienta para realizar integración continua, se decidió utilizar **TravisCI**, y como herramienta de automatización **Gradle**.

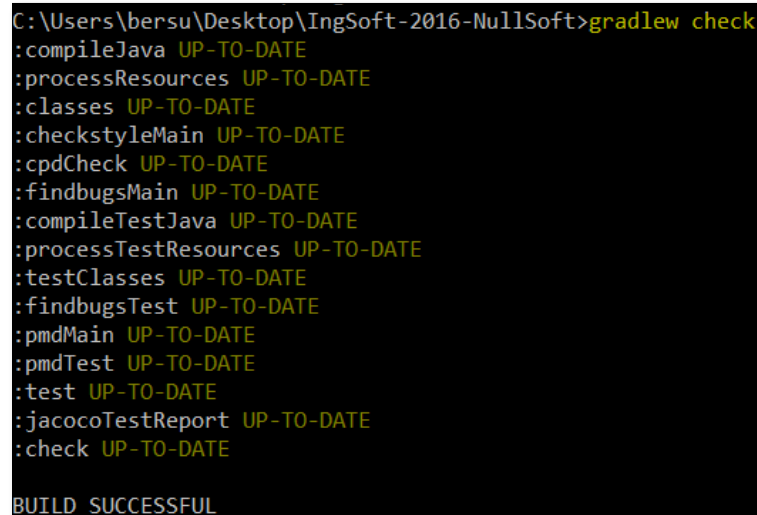
A esta la herramienta de automatización se la configuraron los siguientes plugins:



```
build.gradle: Bloc de notas
Archivo Edición Formato Ver Ayuda
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'pmd'
apply plugin: "de.aaschmid.cpd"
apply plugin: 'findbugs'
apply plugin: "jacoco"
apply plugin: "wrapper"
```

En todos los plugins el build falla si la cantidad de errores supera la cantidad que había en el build anterior. Además de esto si hay problemas en la compilación, o con los test automatizados también fallara el build.

Para hacer el chequeo y correr todos los scripts antes mencionados se debe correr el comando “**gradlew check**” en el servidor de integración continua.



```
C:\Users\bersu\Desktop\IngSoft-2016-NullSoft>gradlew check
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:checkstyleMain UP-TO-DATE
:cpdCheck UP-TO-DATE
:findbugsMain UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:findbugsTest UP-TO-DATE
:pmdMain UP-TO-DATE
:pmdTest UP-TO-DATE
:test UP-TO-DATE
:jacocoTestReport UP-TO-DATE
:check UP-TO-DATE

BUILD SUCCESSFUL
```

En la imagen se puede observar el resultado de un build exitoso.

## CONCLUSIONES

La experiencia obtenida en el transcurso del cursado de la materia Gestión de la Calidad de Software nos permitió tener una noción práctica de cómo mantener un nivel de aseguramiento de la calidad, procurando en todo momento que el costo e impacto sean mínimos.

Pudimos utilizar herramientas de desarrollo de software complementarias que se utilizan en entornos productivos reales con el objetivo de trabajar ordenadamente, automatizando procesos y mejorando la calidad del código.

Esta oportunidad también nos permitió desenvolvernos de manera colaborativa con compañeros y futuros colegas de la carrera.

Sin lugar a dudas, las prácticas y conocimientos adquiridos los pondremos en uso en el transcurso de nuestra actividad como profesionales.